Genetic Synthesis of Concurrent Code using Model Checking and Statistical Model Checking*

Lei Bu[1], Doron Peled[2], Dachuan Shen[1], and Yuan Zhuang[1]

[1] State Key Laboratory of Novel Software Technology, Nanjing University, China
[2] Department of Computer Science, Bar Ilan University, Israel

Abstract. Genetic programming (GP) is a heuristic method for automatically generating code. It applies probabilistic based generation and mutation of code, combined with a "natural selection" principle, using a fitness function. Often, the fitness is calculated based on a large test suite. Recently, GP based on the comprehensive model checking verification method was used for synthesizing correct-by-design concurrent code from a temporal specification. A deficiency of model checking based GP is that it uses a rather limited number of fitness values, based on a small number of modes for each verified specification property (e.g., satisfies, does not satisfy a given property). Furthermore, the need to apply model checking on many candidate solutions using the genetic process makes it prohibitively expensive to use an off-the-shelf model checker such as Spin: the repeated invocation of such a tool, compiling the code for a new candidate solution and running it, can render the performance of this approach several orders of magnitude slower than using an internal model checking. We describe here the use of a combination of model checking and statistical model checking for calculating the fitness required by GP, where the fitness is calculated based on statistical model checking (based on Plasma), which can give smoother fitness values and a light use of model checking (based on Spin) for synthesizing concurrent programs.

1 Introduction

The classical approach for synthesis of interactive systems from temporal specification uses automata and game theory [21]. Synthesis of distributed or concurrent programs from temporal specification is in general undecidable [22]. This calls for the use of heuristic methods. In particular, genetic programming based on model checking [13] employs a powerful heuristic search in the state space of candidate programs, which can be controlled and adjusted by an intelligent user.

Genetic programming (GP) is an automatic method for generating code. It is based on beam search, i.e., a search that maintains in each generation a set of objects, rather than a *single* object. The search attempts to improve the quality of candidates from one generation to another, with mutual influence between candidates. Candidates propagate from one generation to the next one with probability based on their *fitness* value, which

* The research in this paper was partially funded by an ISF-NSFC grant "checking cyber physical systems" (ISF award 2239/15, NSFC No. 61561146394). The first author was also partially funded by a National Natural Science Foundation of China award No. 61572249.

is an estimation on how close the candidate is from a correct solution. In addition, it uses the genetic operations of *mutation*, i.e., making small random changes to a candidate, and *crossover*, i.e., combining elements of two candidates. Since GP does not use backtracking, the only possibilities to deal with a failed search is to start with a new random seed or to try searching with a different fitness function.

Classical genetic programming is based on calculating the fitness function with respect to a large training set of test cases. Recently, using model checking for calculating the fitness function was studied in [12–16]. Model checking is a comprehensive approach for checking correctness, hence its use provides a greater assurance of the correctness of the code than testing. On the other hand, as there are usually a small number of correctness properties, using model checking provides a very small set of fitness values. Additional fitness levels can be provided, e.g., "some executions satisfy a property" or "the property is satisfied with probability 1". However, even then, the fitness landscape is far from being smooth [7], which may sometimes limit the ability of the genetic search to converge. Another difficulty is that model checking is intractable; adding the level of "satisfied with probability 1" is even further complex, as it calls for determinization of the checked property automata. This can involve another exponential blowup, on top of the one that can be incurred when translating LTL to automata.

Model checking based GP requires performing model checking for the many different candidate programs generated during the process. Hence, it benefits greatly from having a dedicated model checker that is implemented within the genetic programming tool. Without it, the use of an off-the-shelf model checking tool like Spin [11] would be several orders of magnitude slower. This motivates using alternative ways of checking the fitness of candidates such as randomized testing and statistical model checking. However, these methods only provide a limited assurance about the correctness of the generated code.

In this work, we suggest to use statistical model checking (SMC) [18, 23] to replace part of the work that is done by model checking. SMC is simulation-based solution, which is less time and memory intensive than classical model checking. The procedure of SMC is to generate enough sample execution paths for the system and then use methods like statistical hypothesis testing to decide whether the system satisfies the given property or not. Therefore, SMC can estimate the probability that a system satisfies a given property. Compared with classical model checking, which can tell only yes and no, this probability measurement gives more smooth answer about how the model satisfies a given property, which can assist in convergence of the process. Typical tools for SMC include Plasma [19] and Uppaal [3].

Applying SMC, which is based on finite executions, we immediately experience several inherent obstacles. The statistical sampling of the executions are limited to finite length, hence, the correctness of the generated programs is not guaranteed by the statistical evaluation. In particular, some properties may fail in very few executions (rare events), which may be missed in the statistical evaluation. Conversely, properties that hold for long or infinite execution sequences may not be manifested during some of the finite executions that are checked. This suggests using a combination of SMC and model checking to achieve the benefit of both methods.

We present here the challenges in replacing model checking by sampling based SMC for GP, and show some measures we took to tackle these problems. Our approach combines SMC with light use of model checking, based on the Spin model checker, performing at the later stages of the genetic process.

2 Genetic Programming

During the 1970s, Holland [10] established the field known as *Genetic Algorithms* (GA). Individual candidate solutions are represented as fixed length strings of bits, corresponding to chromosomes in biological systems. Candidates are evaluated using a *fitness* function. Fitness approximates the distance of the candidate from a desired solution. Genetic algorithms evolves a *set of candidates* into a successor set. Each such set forms a *generation*, and there is no backtracking. The different candidates in a single generation have a combined effect on the search, as progress tends to promote and improve the candidates that are better according to the fitness function and subsequently improve the fitness average across subsequent generations. Candidates are usually represented as fixed length strings. They progress from one generation to the next one according to one of the following cases:

- *Reproduction.* Part of the candidates are selected to propagate from one generation to the subsequent one. The reproduction is done at random, with probability relative to the fitness value or to the relation between the fitness of the selected individual and the average of fitness values in the current generation.
- *Crossover.* Some pairs of the candidates selected for reproduction are chosen, with some given probability, to be combined using the crossover operation. This operation takes parts of bitstrings from two parent solutions and combines them into two new solutions, which potentially inherit useful attributes from their parents.
- *Mutation.* This operation randomly alters the content of small number of bits from candidates selected for reproduction (this can also be done after the crossover). One can decide on mutating each bit separately with some probability.

The process of selecting candidates from the previous generation and deciding whether to apply crossover or mutation continues until we complete a new generation. All generations are of some predefined fixed size N. This can be, typically, a number between 50 and 500. Genetic algorithms thus perform the following steps:

1. Randomly generate N initial candidates.
2. Evaluate the fitness of the candidates.
3. If a satisfactory solution is found, or the number of generations created exceeds a predefined limit (say hundreds or a few thousands), terminate.
4. Otherwise, select candidates for reproduction using randomization, proportional to the fitness values and apply crossover or mutation on some of them,
   again using randomization, until N candidates are obtained.
5. Go to step 2.

If the algorithm terminates unsuccessfully, we can restart it with a new random seed, or change the way we calculate the fitness function.

Genetic programming, suggested by Koza [17], is a direct successor of genetic algorithms. In GP, each individual organism represents a computer program. Programs are represented by variable length structures, such as trees (see Figure 1) or a sequences of instructions. Representation of code uses syntax trees, as shown in Figure 1. It is quite easy to transfer between a program and a syntax tree and vice versa. These trees are well typed. Each node is classified as *code*, *Boolean*, *condition* or *expression*. Leaf nodes are variables or constants, and other nodes have successors according to its type. For example, a while node (of type *code*) has one successor of type *Boolean* or *condition* and one successor of type *code* (for the loop body); the *Boolean* node and has two successors that can be of type *Boolean* or *condition*, and a *condition* node $<$ has two successors of type *expression*. The genetic operations need to respect these (and possibly further) types, e.g., *expressions* cannot be exchanged with *Booleans*.

Crossover is performed by selecting subtrees on each of the parents, and then swapping between them. This forms two new programs, each having parts from both of its parents. There are several kinds of mutation operations. In *replacement* mutation, one picks at random a node in the tree, which roots a subtree. Then one throws away this subtree and replace it with a subtree of the same type, which is generated at random. In Figure 1, the rightmost leaf node was chosen, which is marked with double ellipse. The subtree consists of this single node, represents the constant 1. Thus, it needs to be replaced with another expression, built at random. A new subtree was randomly generated, consisting of two nodes, representing the expression $a[0]$. In *insertion* mutation, a new node of the same type as the selected subtree is generated and is inserted just above it (type permitting); then one may need to complete the tree by constructing another descendant of the newly inserted node. For example, if we select an expression and insert above it a node that corresponds to addition $+$, it can be made one of the descendants to be summed up (say left), but need to complete the tree with a new (right) descendant. The *reduction* mutation has the opposite effect of *insertion*: the selected node is replaced with one of its offsprings (type permitting). In *deletion* mutation we remove the selected subtree, and recursively update the ancestors to make the program syntactically correct.

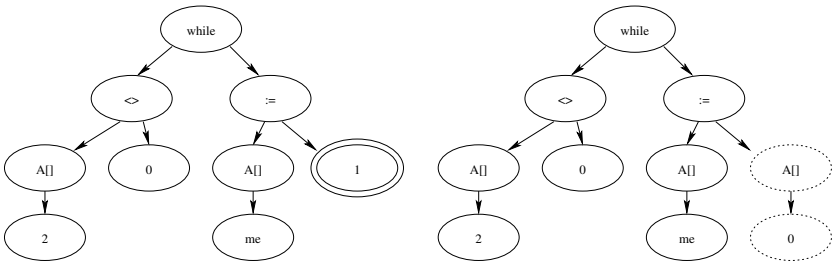

Fig. 1. Mutation on a syntax tree

Syntax trees are not limited to a fixed size. Therefore the candidates can shrink or grow after mutation and crossover. In GP, there is actually a tendency of candidates to *bloat* with unnecessary code, for example, an assignment such as $a[1] := a[1]$. The countermeasure for this, called *parsimony pressure*, is to provide a (small) negative value to the fitness function corresponding to the length of the code. As a consequence, the resulting solutions are not expected to have a perfect fitness value, but instead they need to pass all the tests/verifications performed.

3 GP based on Model Checking and Statistical Model Checking

We want to employ GP to synthesize concurrent programs from given temporal specifications. We use LTL syntax for describing the behavior of desired programs. The input includes, besides the temporal specification, also a *configuration*, which restricts the parameters of the desired solution. The configuration can restrict the depth of the generated syntax trees, the variable used, the allowed arithmetic and Boolean operators, and the number of processes. It can also contain a template that restricts the code, e.g., dictates that the code is embedded within a fixed loop or contains some fixed parts of code. The template has several uses:

- Using the template we can guarantee part of the behavior of the sought after code, simplifying the specification.
- We use linear temporal logic, which is limited to assert on *all* the executions of the code. On top of that, we can use the template to force checking different cases, providing some expressive power of branching temporal logic such as CTL [2]. Furthermore, it also provides a testbed environment with *uncontrolled* actions, where the code needs to behave under all the interactions with it. This provides some expressive power of game logics [5].
- The template can be used to limit the state space of the search, e.g., suggesting that a solution will start with an assignment, or that it is embedded in a main loop. This can reduce the complexity of the genetic search and improve the chance and speed of coverage.

In [12], GP based on model checking was described and experimented with. The fitness function was calculated based on model checking results. Using model checking (which is an exhaustive check) instead of testing to calculate the fitness function for GP allows a more reliable evidence of the correctness of the code. On the other hand, model checking is computationally expensive. In [13–16], it was observed that the number of specification properties is rather small, which creates a small number of fitness values. Therefore, a few intermediate levels were added on top of the obvious *satisfies*/*does not satisfy* verdicts; in particular, levels such as *sometimes satisfies* and *satisfies with probability 1*.

3.1 A Running Example

As a running example, we look at synthesizing a solution for the well known *mutual exclusion* problem. Solutions for mutual exclusion from temporal specification were

synthesized using GP, where the fitness function is based on model checking [13, 14]. The configuration provided dictates the following structure:

```
p1: While true do        p2: While true do
  nonCrit1                 nonCrit2
  preCS1                   preCS2
  CS1                      CS2
  postCS2                  postCS2
end while                end while
```

The label nonCrit*i* represents the actions of the process p*i* outside the critical section. It can actually be fixed as empty code. The label CS*i* represents the critical section, which both processes want to enter a finite or unbounded number of times. It is not part of the synthesis task, and can also be represented by trivial code entering and exiting. The critical section is controlled by the code that will be synthesized instead of preCS*i* and postCS*i*. We require the following Linear Temporal Logic properties:

Safety: $\Box\neg(\mathsf{p0}\ \mathit{in}\ \mathsf{CS1} \wedge\ \mathsf{p1}\ \mathit{in}\ \mathsf{CS2})$, i.e., there is no state where the program counters of both processes are in the critical section simultaneously.

Liveness: $\Box(\mathsf{p}i\ \mathit{in}\ \mathsf{preCS}i \rightarrow \Diamond \mathsf{p}i\ \mathit{in}\ \mathsf{CS}i)$, i.e., if a process wants to enter the critical section, then it will eventually do so.

In general, a solution that restricts both processes to alternate in entering the critical section would satisfy the temporal conditions. However, if a solution that necessarily alternate between the process in entering their critical sections would also satisfy these conditions. Then, if one of them cease to try to enter its critical section, the other one can get blocked.To eliminate this problem, we made use of the configuration: the variables W*i* in the configuration are used to control whether processes want to keep entering the critical section. This part of the code is fixed and not subject to synthesis. The program now has to satisfy the liveness when $(\mathsf{W1} \wedge \mathsf{W2})$, or $(\neg\mathsf{W1} \wedge \mathsf{W2})$, or $(\mathsf{W1} \wedge \neg\mathsf{W2})$.

Note that the configuration assures that the duration of the critical sections CS*i* are finite. Hence there is no need to require that $\neg\Diamond\Box \mathsf{p}i\ \mathit{in}\ \mathsf{CS}i$. The lack of jumps (goto) in the code alleviates us from requiring that we cannot move from one part of the above constructs to the middle of another part.

3.2 Replacing Model Checking with Statistical Model Checking

Due to the two deficiencies of the use of model checking in genetic programming mentioned: complexity and lack of smoothness of the fitness value, we were motivated to replace part of its use by statistical model checking. In particular, we generate for each GP candidate solution a (large) set of (pseudo) random executions; we check if these executions satisfy related specification properties.

The fitness function used in GP needs to be rather smooth in order to provide good convergence, and the statistical evaluation can provide multiple levels. Statistical evaluation may also be more affordable for some intricate synthesis problems. The simplicity of statistical methods is even further apparent for real time or cyber-physical systems.

Another advantage that statistical model checking has over model checking is that it can be used for parametric systems and systems with infinite state space, where model checking has limited use for these applications.

For using statistical model checking over *finite* prefixes, we form a set of bounded temporal properties over *finite* prefixes of executions that are related to the original LTL properties over infinite sequences. We use the same syntax as LTL over infinite sequences, but the interpretation over a finite sequence is if its last state is repeated forever. Safety properties can be migrated directly to finite prefixes: by definition [1], a safety property is violated when there is a finite prefix that does so. We use the notation $\blacklozenge\varphi$ to denote that φ holds at the end of the finite sequence (if the sequence has length k, then $\blacklozenge$ can be replaced by k successive $\bigcirc$s). Nevertheless, a finite prefix may present only partial information, and the property may be violated or satisfied only in a longer prefix. The properties over finite prefixes that correspond to the original properties may provide support to the case that the original properties hold for the infinite sequences. For example, instead of the liveness property, we may use a property that a process enters its critical section some fixed amount of times. The larger this number is, the more we are convinced that the liveness property holds. However, a large number will only be manifested in a long prefix. We pick up these related properties over finite prefixes according to our intuition (we may fine tune them if the genetic process fails).

At the moment we do not have a way of obtaining these related properties *automatically* from the original LTL properties, and this can be the subject of further research (e.g., using genetic co-evolution or learning). Nevertheless, we do not expect the synthesis of concurrent programs to be completely automatic, as it was shown to be undecidable [22]

We will illustrate the choice of related properties over finite prefixes for the running example. Suppose that we decide to check n executions, each one of them is limited to a length of k. We can fine tune the parameters n and k on several test runs to see what works. We can also try to estimate the size (number of states) of the desired solution to provide such parameters where errors will be found with high probability [9]. However, the latter may require an impractical set of checked executions.

We split the safety and liveness properties into a set of bounded properties. This set of formulas take care of the following cases:

- M. Mutual exclusion holds: $\rho_M = \Box\neg(\mathsf{p0}\ \mathit{in}\ \mathsf{CS1} \wedge\ \mathsf{p1}\ \mathit{in}\ \mathsf{CS2})$.
- B. The case that both processes want to enter the critical section. We enforce that by setting $(\mathsf{W1} \wedge \mathsf{W2})$. We add two counters enter_1 and enter_2 to indicate the times that each process enters the critical section. Out of which we have:
  - B_1. Both processes succeed entering the critical section multiple times: $\rho_{B_1} = \blacklozenge(\mathit{enter}_1 > 1 \wedge \mathit{enter}_2 > 1)$.
  - B_2. One process enters the critical section multiple times and the other only once: $\rho_{B_2} = \blacklozenge((\mathit{enter}_1 > 1 \wedge \mathit{enter}_2 = 1) \vee (\mathit{enter}_1 = 1 \wedge \mathit{enter}_2 > 1))$.
  - B_3, Only one process succeeds in entering, or both enter exactly once: $\rho_{B_3} = \blacklozenge(\mathit{enter}_1 + \mathit{enter}_2 \geq 1 \wedge 0 \leq \mathit{enter}_1 \times \mathit{enter}_2 \leq 1)$.
  - B_4. Both processes do not succeed in entering their critical section $\rho_{B_4} = \blacklozenge(\mathit{enter}_1 + \mathit{enter}_2 = 0)$.

– O. Only one process wants to enter, when forcing $(\mathsf{W1} \wedge \neg \mathsf{W2})$. Out of which we have:
  - O_1. The process succeeds entering the critical section multiple times: $\rho_{O_1} = \blacklozenge(enter_1 > 1)$
  - O_2. The process succeeds entering the critical section only once: $\rho_{O_2} = \blacklozenge(enter_1 = 1)$
  - O_3, The process does not succeed entering the critical section: $\rho_{O_3} = \blacklozenge(enter_1 = 0)$

We mark the SMC probabilities (as estimated by an SMC tool, or just the portions of executions satisfying each property among the randomly generated test cases) of the model satisfying these given properties by P_M, P_{B_1}, P_{B_2}, P_{B_3}, P_{O_1}, and P_{O_2} respectively. We can base the fitness functionbased on the above parameters. This function should give different weights to the various probabilities of the tested executions satisfying the different properties above.

We have the following coefficients, which can be assigned various values between 0 and 1:

– α multiplies P_M, the probability that the model satisfies the safety property.
– $\beta_1, \beta_2, \beta_3$ multiply P_{B_1}, P_{B_2}, P_{B_3}, the probability that the model satisfies ρ_{B_1}, ρ_{B_2} and ρ_{B_3}, respectively.
– γ_1,γ_2 multiply P_{O_1}, P_{O_2} the probability that the model satisfies ρ_{O_1},and ρ_{O_2}, respectively[3].

We enforce that $\beta_3 < \beta_2 < \beta_1$, $\gamma_2 < \gamma_1$. A possible fitness function is

$$(\alpha \times P_M + \beta_1 \times P_{B_1} + \beta_2 \times P_{B_2} + \beta_3 \times P_{B_3} + \gamma_1 \times P_{O_1} + \gamma_2 \times P_{O_2}) \times 100$$

We normalize fitness to be between 0 and 100 by requiring that $\alpha + \beta_1 + \gamma_1 = 1$.

3.3 Problems and solutions in using SMC for fitness function

We need to pay attention to some issues in transforming SMC probabilities into fitness results. We will first list the difficulties, and then suggest some solutions.

Limited distinction of the probabilistic approach. Although providing multiple fitness levels, SMC based fitness function is only a rough and priori heuristic estimate. In particular, it is hardly reasonable to assume that a solution that has 75% of its sampled prefixes satisfy some properties is uniformly better than one in which 85% of the sampled prefixes satisfy them. (However, the use of stochastic selection of candidates for propagation by the genetic programming algorithm, where the given fitness only affects the *probability* of selecting the candidate, rather than directly selecting the best fitted ones, somewhat smoothens out the difference between these similar cases.)

False positives: Failure of properties that appear as rare events. The executions where an error is demonstrated may be rare; in which case one may need a lot of experiments and would, by chance, not catch the bad executions. For mutual exclusion, the

[3] The coefficients for ρ_{B_4} and ρ_{O_3} are both 0, as these cases are failed.

processes *may* enter the critical section simultaneously, but on many executions they just independently enter and then exit, where the simultaneous stay within the critical section is not manifested on the selected random prefixes.

False negatives: Negative bias due to scheduling. Another problematic situation is where some liveness properties would not show up on a substantial number of prefixes due to scheduling. In a particular finite execution, a process may fail to enter the critical section since the other process is scheduled more frequently, although it could do so in a longer prefix or under a different scheduling.

Fairness. Many solutions of the mutual exclusion are based on some fairness assumption [20]; there, without allowing both processes ample opportunities to progress, the liveness will not hold. In particular, this is the case for the classical *Dekker* solution for mutual exclusion, presented by Dijkstra [4]. However, fairness is defined over infinite executions, and SMC checks only finite ones.

In order to tackle the above issues, that stem from the randomness and finiteness of the checked prefixes, we used a combination of the following ideas.

Extending the measurements. Depending on the checked property, we may want to extend the measurements. For example, for the safety property, we may want to check more executions to increase the probability that we find the violation. For the liveness property, we may want to use longer executions to diminish the effect of unfair scheduling. These parameters are adjusted after some initial failures to synthesize correct solutions.

Using combination of cases. Because we cannot rely on fairness, and our tested sequences are finite, we should learn about the satisfaction of a property from observing the *combination* of the random checks. Take for example the case where we want to check that a process is not prohibited from entering the critical section. There may exist some prefixes where it fails to do so. However, if in a substantial amount of the executions (say, 70%), it succeeds in entering the critical section, this can be used as evidence that the failure in the minority of the executions is due to unfavored scheduling. Then, given a certain threshold, we may apply "majority rules" to conclude that the liveness property holds. Accordingly, we may decide that when at least 70% of the B executions are satisfying ρ_{B_1}, the fitness treats all the executions in B as if they all satisfy ρ_{B_1}. Suppose the probability that the execution falls in B is P_B, if $P_{B_1} > P_B \times 0.7$, then we can use a simpler fitness function $(\alpha \times P_M + \beta_1 \times P_B + \gamma_1 \times P_{O_1} + \gamma_2 \times P_{O_2}) \times 100$.

However, this is not the only possible conclusion for this measurement: it may not be the discrimination of the scheduling that makes a process fail to enter its critical section, but instead there may be some scheduling that subsequently prevents the entrance to the critical section. Such a situation of multiple possible conclusions from the same statistical experiments can be resolved by the combined use of both majority rules *and* the light use of model checking (see below); model checking will catch such rare event errors that may otherwise not affect the fitness function.

Biasing the probabilistic selection. If we identify cases that may happen rarely, we can use biasing of the different choices in order to inspect them closer. For example, since catching violation of the safety property may be rare, we can reduce the probability of transitions that *exit* the critical section in favor of transitions of the other process that is outside the critical section. In essence, we are "waiting" for the other process to enter

the critical section. For promoting liveness and providing more "fair" scheduling, we can decrease the probability of a transition of some process to be selected relative to the number of states where the other process has been waiting.

Light use of model checking as certification or as part of the fitness. When candidates that receive very high fitness values are produced, in late generations, model checking can be used to certify that they indeed satisfy the desired properties. One can apply model checking sparingly, late on the genetic process, on candidates with already very high fitness value. We then may integrate the result of the model checking into the fitness and allow it to participate in additional generations.

Checking ultimately periodic executions. We can replace checking finite executions by ultimately periodic ones. This can be done as in [6]. However, checking ultimately periodic sequences is more expensive than checking finite prefixes, as states on a sequence need to be hashed in order to detect cycles. This part was not implemented in our prototype.

4 Experiments

We performed experiments on using model checking and statistical model checking for calculating the function of genetic programs. For each combination, we tried synthesizing code for mutual exclusion, round robin and the dinning philosophers.

For each of these problems we have run experiments with SMC using both Plasma [19], and running our own statistical evaluation tool. Plasma uses Approximate Probabilistic Model Checking (APMC) [8] to provide a controlled accuracy on the statistical results[4]. For accelerating the performance, we have implemented an ad-hoc statistical evaluation we denote by N-SMC (for *naive SMC*) that simply selects a given number of finite sequences and calculates the ratio of executions that satisfy a given property.

The model checking is performed by Spin. Spin works here as separate software interfacing with ours, which needs to prepare its own (multiple) files and performs compilation on each candidate it checks, in order to make the verification; each activation of Spin by our code is slower than the statistical evaluations we make per candidate, hence we apply it sparingly. The Spin model checker is invoked when the fitness function reaches a high threshold, which is set to 98 in the experiments. If model checking fails, we continue the genetic process (since the failed candidate solutions may still contain good "genetic material" so we can proceed from this point based on the SMC fitness calculations).

4.1 Synthesis of Solutions for Mutual Exclusion

The first set of experiments we conducted is to use GP to synthesize solutions of mutual exclusion. Without using Spin in the last stage to do the certification, our implementation can generate dozens of solutions that reach the highest fitness value easily. For

[4] The configuration of running Plasma in our experiment includes the approximation threshold $\varepsilon = 0.05$, and the confidence threshold $\delta = 0.01$. Please refer to [8] for detail explanation of the meaning of these parameters.

example, three representative solutions (a), (b) and (c) are shown below. The processes are symmetric, hence we represent in each case only one of them ($p1$).

```
While W1 do
 v[me]=1
 While (v[2]!=me) do
  v[2]=0
  if(v[other]!=me)
    v[2]=1
 end while
 CS
 v[2]=other
end while
   (a)
```

```
While W1 do
 v[me]=1
 While (v[2]==other) do
  v[2]=1
  if(v[other]!=other)
    v[2]=0
 end while
 CS
 v[2]=other
end while
   (b)
```

```
While W1 do
 v[me]=1
 While (v[other]!=0) do
   While (v[other]==other) do
     v[me]=0
   end while
   v[me]=1
 end while
 CS
 v[me]=0
end while
   (c)
```

In the random simulations, both the processes show no violation of the mutual exclusion, starvation or deadlock. However, if we investigate the two solutions (a) and (b), we can find that they fail to satisfy the safety requirement. In some execution sequences, two processes can enter the critical section at the same time. Actually, when we simulated solution (a) for 10000 times, we could observe only 139 times failure to satisfy the safety requirement. The unsafe scenario happened even fewer times in scenario (b): 4 times in 100000 simulations.

On the other hand, solution (c) does not satisfy the liveness property. Actually, this solution represents the scenario where only if both processes want to enter the critical section infinitely, then the liveness is satisfied; however, if one process decides to stop, then the other process will be blocked forever. These examples confirm the problems we raised in Section 3.3, where one may need a lot of experiments and may, by chance, even then not catch the bad rare events. This leads us to the next experiment, where we used model checking as certification in the last generation of the genetic process.

When candidates that received very high fitness values, 98 in the experiments, were produced, we used model checking to certify whether they indeed satisfy the desired properties. The model checker used is Spin [11]. We implement an automatic generator to translate the solution generated by GP into the modeling language PROMELA of Spin. Then, we used Spin to check all the solutions with fitness value above the threshold against the requirements. If the model checking confirmed correctness, the procedure was stopped. Otherwise, we continued the GP process until the limit on the number of generations has been reached.

With the help of Spin, we get a set of correct solutions. One correct solution we generated is (d) below. This is a perfect solution that shares a similar structure with Dekker's algorithm. Another representative solution (e) is similar to Peterson's algorithm. The difference between (e) and (d) is that (e) allows Boolean operators *and* and *or* into the conditions.

```
While W1 do
 v[me]=1
 While (v[other]==1) do
  While (v[2]!=other) do
   v[me]=0
  end while
  v[me]=1
 end while
 CS
 v[me]=0
 v[2]=me
end while
```
(*d*)

```
While W1 do
 v[me]=1
 v[2]=me
 While((v[other]!=0) && (v[2]==me)) do
 end while
 CS
 v[2]=other
 v[me]=0
end while
```
(*e*)

4.2 Synthesizing Solutions for Round Robin Scheduling

After mutual exclusion problem, we check our system on another scheduling example. In this example, we have three processes p_0, p_1 and p_2 that share a critical section. However, in this version, they need to enter it in round robin order, that is: p_0 before p_1, then p_2, and repeating that order with p_0, etc. The processes always want to enter the critical section (there is no flag W*i* that restricts a process from wishing to enter). A trivial solution is that the processes would use a turn variable with three values, 0, 1 and 2, and each process will enter only when turn points to it and then increment it modulo 3. However, to make things less trivial, we require that we use only Boolean variables.

We allow solutions that are asymmetric in the sense that different values will be assigned in different processes. To allow that but still generate one candidate that will be concretized into three processes, we introduced to the generated process template as a syntactic construct an assignment statement of the form v[i]=$b_0b_1b_2$, $b_i \in \{0,1\}, 0 \leq i \leq 3$. This dictates that for the actual process p_i, the concretized statement will be v[i]=b_i. The variables which can show up in the solution are v[0] to v[3], and also v[me], v[other1], and v[other2]; for each process p_i, me will be concretized by i, other1 will be concretized as $(i+1)$ *mod* 3, and other2 will be concretized as $(i+2)$ *mod* 3.

We adapted the SMC based fitness function to comply with three processes and the round robin specification. Our GP based synthesizer generated several solutions similar to the following solution (*f*).

```
While true do
  v[3]=010
  While(v[me]==001) do
    v[3]=101
    v[3]=010
  end while
  CS
  v[me]=001
  v[other1]=101
end while
```
(*f*)

```
While true do
  While(v[me]==001) do
  end while
  CS
  v[me]=001
  v[other1]=101
end while
```
(*g*)

We can see that each process in solution (*f*) refers to v[me] and v[other1]. There are also assignments to v[3] among the statements. However, as v[3] is not used in any conditions at all, such statements can be safely removed (this is done here manually to demonstrate the solution), resulting in solution (*g*). As explained above, the generation of useless but harmless statements is common in GP and is called parsimony. A standard solution for tackling bloating is to apply some small negative fitness value related to the size of the code, which we did not do at this point.

Let us concretize the solution of the three process p_0, p_1, and p_2 to (g_0), (g_1), and (g_2) respectively. Observe that only three variables are used in the solution, which makes this solution simple and elegant.

```
While true do
  While(v[0]==0) do
  end while
  CS
  v[0]=0
  v[1]=1
end while
```
(g_0)

```
While true do
  While(v[1]==0) do
  end while
  CS
  v[1]=0
  v[2]=0
end while
```
(g_1)

```
While true do
  While(v[2]==1) do
  end while
  CS
  v[2]=1
  v[0]=1
end while
```
(g_2)

4.3 Synthesizing Solutions for Dining Philosopher

The third experiment we do is the dining philosopher problem, which is a well-known concurrent problem. Several philosophers sit in a round table. A philosopher can take the fork on his right or the one on his left as they become available. If a philosopher wants to eat, he must have both left and right forks and if he finishes eating, he needs to put the forks back and these forks will be available again. The problem is to design a concurrent algorithm such that no philosopher will starve.

To support this problem, we extend the basic variable library with semaphore variables, and also add semaphore-related operations such as wait and signal into the expression library. Two representative solutions generated by our method are shown below in (*h*) and (*i*). It is interesting to notice that although there are different kinds

of statements like while loop, if condition, and variable assignments in the statement library, the automatically generated solutions are composed by only manipulation of semaphores. In our solutions, each philosopher/process waits for the free of the globally shared semaphore "mutex" firstly, then take both the forks by manipulating semaphores "left" and "right", which will be translated as $i \ mod \ m$ and $i+1 \ mod \ m$, for the ith philosopher in m. After finishes eating, the dining philosopher will free the semaphore of the holding forks and also the globally shared mutex semaphore.

Both solutions (h) and (i) can pass the verification of Spin. The difference is that, in (h), when one philosopher is eating, all the other philosophers are blocked. While (i) is more effective, only the philosophers share the forks with the dining ones will be blocked.

```
While W1 do
  think
  wait(mutex)
  wait(right)
  wait(left)
  eat
  signal(left)
  signal(right)
  signal(mutex)
end while
```

(h)

```
While W1 do
  think
  wait(mutex)
  wait(right)
  wait(left)
  signal(mutex)
  eat
  signal(right)
  signal(left)
end while
```

(i)

4.4 Performance Evaluation

In this section, we report the performance data of the method presented in this paper. We compare the performance of the previous pure model checking guided GP synthesis, the SMC (Plasma) guided GP synthesis and also our own native statistical analysis based version. The corresponding data are marked as SMC and N-SMC respectively. Meanwhile, the data for mutual exclusion, round robin and dining philosopher problems are marked as ME, RR, and DP respectively.

In the experiment, we have 100 seeds for each generation, and if the GP does not generate a correct solution in 2000 generations, we abandon the current GP search. We try to execute each problem 100 times. The time limit is a week. We record the average time each execution takes, the number of successful executions which generate perfect solution, the average number that the GP calls model checker per execution, and the average generation number per execution. The performance data is given below in Table.1.

As the N-SMC built-in tool basically implements a simplified methods of SMC, we can see the success ratio of N-SMC for each problem aligns with SMC, but performs almost two orders of magnitude faster. However, SMC is an of-the-shelf tool, same as Spin, and the idea of using N-SMC was to diminish the effect of having most of the overhead due to the invocation and repeated use of the tool. In fact, given that Plasma was used in this mode, its use should be considered very efficient. Moreover, the slight

Table 1. Performance Data

Problem	Method	Total Executions	Average Time	Success Executions	Success Rate	Average Spin Call	Average Generations
ME	SMC	21	7h53m	0	0	280	2000
	N-SMC	100	341.4s	15	15%	378.3	1754
RR	SMC	22	7h25m	8	36.4%	502	1751
	N-SMC	100	479.4s	41	41%	436.3	1389
DP	SMC	11	15h23m	9	81.8%	1178	687
	N-SMC	100	34m6s	65	65%	1287	116

need to invoke Spin (in both modes), again, as an external tool, in the later stage, did not make the entire synthesis process prohibitively expensive.

This experiment gives us strong belief that our proposed SMC guided genetic synthesis method can be applicable and efficient in concurrent code synthesis with the help of a built-in implementation of SMC and model checking procedures. It can generate a correct solution with high success ratio within reasonable time overhead. Moreover, allowing SMC tools such as Plasma or Uppaal [3], and model checking tools such as Spin to be integrated into a GP tool would make the genetic synthesis both efficient and powerful.

We do not compare our experiments directly to the results in [13–16] since there, an internal tool for performing model checking was used, which was tailored to provide additional levels besides the yes/no (+counterexample) that standard model checkers provide. This presents several difficulties. First, an internal model checker needs to be implemented, which can hardly compete with the breadth of a tool like Spin. The second problem is that even with the additional levels (never satisfies a property, almost always satisfies a property etc.), the fitness function may not be smooth enough in some cases.

5 Conclusions

We described here the use of genetic programming based on statistical model checking for synthesizing concurrent code from its temporal specification. Using statistical model checking for defining the fitness function has several advantages over using model checking. In particular, it can be more efficient, can be used in domains where model checking is not applicable, and can provide a smoother function, which helps to converge. We presented different ideas and parameters for defining statistical based fitness function.

We implemented the presented ideas and conducted experiments of synthesizing concurrent code, from which we have learned several lessons. One of the main lessons is that some common properties, such as mutual exclusion or eventual progress, may happen to be quite elusive in a model. This makes the tradeoff between efficiency, where a rather small number of random executions are checked, and reliability, where we use a large set of random samples. This also calls for using model checking to verify the generated solutions.

We used a hybrid approach, where we used statistical model checking for most of the duration of the genetic process, but involved model checking at the later part of the genetic process to certify the potential solution.

Our research on the combination of generic programming, synthesis, statistical model checking and model checking already shows some encouraging results, but also calls for several follow ups. Besides improving our tool (in particular, using built-in verification, as the connection with Spin and Plasma is quite time consuming) there are some interesting theoretical/practical directions. One direction is the use of biasing of the randomized experiments. Finally, we intend to make more experiments in synthesizing code, in particular of timed, probabilistic and cyber physical systems, where statistical approaches, such as statistical model checking, are found to be quite efficient.

References

1. ALPERN, B., AND SCHNEIDER, F. B. Recognizing safety and liveness. *Distributed Computing 2*, 3 (1987), 117–126.
2. CLARKE, E. M., AND EMERSON, E. A. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs* (1981), pp. 52–71.
3. DAVID, A., LARSEN, K. G., LEGAY, A., MIKUCIONIS, M., AND POULSEN, D. B. Uppaal SMC tutorial. *STTT 17*, 4 (2015), 397–415.
4. DIJKSTRA, E. W. Cooperating sequential processes.
5. FINKBEINER, B., AND SCHEWE, S. Coordination logic. In *Computer Science Logic, 24th International Workshop, CSL, 19th Annual Conference of the EACSL, Brno, Czech Republic* (2010), pp. 305–319.
6. GROSU, R., AND SMOLKA, S. A. Monte carlo model checking. 271–286.
7. HARMAN, M., MANSOURI, S. A., AND ZHANG, Y. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv. 45*, 1 (2012), 11:1–11:61.
8. HÉRAULT, T., LASSAIGNE, R., MAGNIETTE, F., AND PEYRONNET, S. Approximate probabilistic model checking. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings* (2004), pp. 73–84.
9. HOEFFDING, W. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association, Vol. 58, No. 301* (1963), 13–30.
10. HOLLAND, J. H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992.
11. HOLZMANN, G. J. *The SPIN Model Checker*. Pearson Education, 2003.
12. JOHNSON, C. G. Genetic programming with fitness based on model checking. In *Genetic Programming, 10th European Conference, EuroGP 2007, Valencia, Spain, April 11-13, 2007, Proceedings* (2007), pp. 114–124.
13. KATZ, G., AND PELED, D. Genetic programming and model checking: Synthesizing new mutual exclusion algorithms. In *ATVA* (2008), vol. 5311 of *LNCS*, pp. 33–47.
14. KATZ, G., AND PELED, D. Model checking-based genetic programming with an application to mutual exclusion. In *TACAS* (2008), vol. 4963 of *LNCS*, pp. 141–156.
15. KATZ, G., AND PELED, D. Synthesizing solutions to the leader election problem using model checking and genetic programming. In *HVC* (2009).
16. KATZ, G., AND PELED, D. Code mutation in verification and automatic code correction. In *TACAS* (2010), pp. 435–450.

17. KOZA, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
18. LEGAY, A., DELAHAYE, B., AND BENSALEM, S. Statistical model checking: An overview. In *Runtime Verification - 1st International Conference, RV 2010* (2010), pp. 122–135.
19. LEGAY, A., AND TRAONOUEZ, L. Statistical model checking of simulink models with plasma lab. In *Formal Techniques for Safety-Critical Systems - Fourth International Workshop, FTSCS 2015, Paris, France, November 6-7, 2015. Revised Selected Papers* (2015), pp. 259–264.
20. MANNA, Z., AND PNUELI, A. How to cook a temporal proof system for your pet language. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983* (1983), pp. 141–154.
21. PNUELI, A., AND ROSNER, R. On the synthesis of a reactive module. In *POPL* (1989), pp. 179–190.
22. PNUELI, A., AND ROSNER, R. Distributed reactive systems are hard to synthesize. In *FOCS* (1990), pp. 746–757.
23. YOUNES, H. L. S., AND SIMMONS, R. G. Probabilistic verification of discrete event systems using acceptance sampling. In *CAV 2002,Copenhagen, Denmark, July 27-31,* (2002), pp. 223–235.